REMARKS

This application has been carefully reviewed in light of the Office Action dated August 18, 2008. Claims 1 to 39, 44, 45 and 53 to 60 are in the application, of which Claims 1 and 53 to 60 are independent. Reconsideration and further examination are respectfully requested.

The Rejection Under 35 U.S.C. § 101

Claim 56 has again been rejected under 35 U.S.C. § 101, as allegedly being directed to non-statutory subject matter. Claim 56 is written in "means-plus-function" language, pursuant to 35 U.S.C. § 112, sixth paragraph. According to the rejection, the various "means" set out in the claim can be implemented via software, codes, programs and/or algorithms, such that in the USPTO's view, the claim as a whole fails to fall within a statutory category and is nothing more than software *per se*.

The rejection continues to be traversed. Three grounds for traversal are asserted: The plain language of the claim itself, the holding of *In re Alappat*, and the statutory mandate of 35 U.S.C. § 112, sixth paragraph.

First, with respect to the plain language of the claim, Claim 56 itself states that it is "an apparatus". An apparatus, even if set out in means-plus-function language, is and has always been statutory under § 101. Withdrawal of the rejection under this basis is respectfully requested.

Second, Applicant repeats and hereby incorporates all prior arguments concerning *In re Alappat*. Like Claim 56 herein, the claim considered by the Federal Circuit in *In re Alappat* was also written in means-plus-function language. The Federal Circuit clearly articulated a line of reasoning as to why such a claim was statutory, and such reasoning applies equally to Claim 56 herein. Moreover, the more recent Federal Circuit decision in *In re Bilski* does not apply here, since the claim in *Bilski* is a method claim and not a means-plus-function claim.

Third, it is respectfully submitted that the USPTO has failed to analyze Claim 56 pursuant to the statutory mandate of 35 U.S.C. § 112, sixth paragraph. Such an analysis was articulated by the Federal Circuit in *In re Donaldson*, 29 USPQ2d 1845 (Fed. Cir. 1994, *en banc*), and described at MPEP § 2181 through § 2186. According to § 112, sixth paragraph, claims written in means-plus-function format do not have unlimited scope; rather, such claims are limited to structure, material or acts described in the specification and equivalents thereof. In rejecting Claim 56, the USPTO has drawn a correspondence between the means-plus-function language of Claim 56 and software *per se*. It is respectfully submitted that such a correspondence has strayed from the statutory mandate of § 112, sixth paragraph, as explained more fully below.

It is worthwhile to recall the precise language of § 112, sixth paragraph, which is applied when interpreting an element written in means-plus-function format:

> "An element in a claim for a combination may be expressed as a means or step for performing a specified function without the recital of structure, material, or acts in support thereof, and such claim ***shall*** be construed to cover the corresponding ***structure, material, or acts*** described

in the specification and equivalents thereof." (Emphasis added.) 35 U.S.C. § 112, paragraph 6.

Clearly, therefore, § 112, sixth paragraph, acts as a limitation and restriction on the scope of a claim. It is not permissible for the USPTO to give unrestricted scope to such a claim. Rather, as explained by the Federal Circuit in *Donaldson*, the USPTO must restrict the scope of the claims to those aspects of the specification that describe structure, material and acts or to equivalents thereof:

> "The plain and unambiguous meaning of paragraph six is that one construing means-plus-function language in a claim must look to the specification and interpret that language in light of the corresponding structure, material, or acts described therein, and equivalents thereof, to the extent that the specification provides such disclosure. Paragraph six does not state or even suggest that the PTO is exempt from this mandate, and there is no legislative history indicating that Congress intended that the PTO should be." *Donaldson* at 1848.

> "[O]ur holding in this case merely sets a limit on how broadly the PTO may construe means-plus-function language under the rubric of 'reasonable interpretation'. Per our holding, the 'broadest reasonable interpretation' that an examiner may give means-plus-function language is that statutorily mandated in paragraph six. Accordingly, the PTO may not disregard the structure disclosed in the specification corresponding to such language when rendering a patentability determination." *Donaldson* at 1850.

Applying this legal framework to Claim 56, it is clear that § 112 provides a limit to the scope of the claim. Claim 56 is limited to "structure, material or acts", or equivalents thereof. The initial burden is on the USPTO, who must identify a correspondence between the various means-plus-function elements of Claim 56 and those

aspects of the specification that describes "structure, material or acts" and equivalents thereof.

Not everything described in the specification necessarily fits the statutory requirement for "structure, material or acts", and it is likewise impermissible for the USPTO to take the position that any and all structure, material or acts are somehow equivalent to those described in the specification.

In addition, it should also be beyond argument that the statutory terminology of § 112, i.e., "structure, material or acts", embraces subject matter that is clearly statutory under § 101.

Here, in its rejection of Claim 56, the USPTO has taken the position that the various "means" of Claim 56 correspond to nothing more than software *per se*:

> "The means as set out in the claim can be implemented via software, code(s), program and/or algorithm [citations to the specification omitted], thus, resulting in a software *per se* claim." Office Action, page 4.

Having drawn such a correspondence, the USPTO is left with only two possibilities. First, if the USPTO is correct in drawing this correspondence between the claimed means and "software *per se*", then pursuant to § 112, sixth paragraph, then "software *per se*" must also be "structure, material or acts" or equivalents thereof. Hence, since the USPTO has taken the position that software *per se* is "structure, material or acts" or equivalents thereof, then the USPTO must also concede that software *per se* is statutory under § 101.

Second, it is possible that the USPTO has drawn its correspondence more broadly than permitted by § 112, sixth paragraph. More precisely, claims written in means-plus-function language are not unlimited in scope. They encompass only structure,

material or acts, or equivalents thereof. Anything that is not a "structure, material or acts", and anything that is not an equivalent of "structure, material or acts", is not literally embraced within the scope of a means-plus-function claim. Under this view, the USPTO is wrong to conclude that software *per se* is one of the permitted interpretations of § 112, sixth paragraph. Rather, and pursuant to § 112, sixth paragraph, the USPTO is permitted only to look to actual "structure, material or acts", as well as to their equivalents.

At its heart, the instant traversal is rooted in the statutory language of § 112, sixth paragraph. Means-plus-function claims are properly limited to "structure, material and acts" and equivalents thereof. All such "structure, material or acts" are statutory under § 101. No other conclusion is possible.

Withdrawal of the § 101 rejection is respectfully requested.

Art-Based Rejections

Claims 1 to 4, 8, 10, 11, 13, 14, 17, 22 to 32, 35, 36, 53, 55 and 57 were rejected under 35 U.S.C. § 103(a) over U.S. Patent 6,125,390 (Touboul) in view of U.S. Patent 6,546,484 (Hirai), and further in view of U.S. Patent Application Publication 2003/0033395 (Sato). Claims depending from Claim 1 were rejected further in view of U.S. Patent 5,696,701 (Burgess) or U.S. Patent 6,167,567 (Chiles).

In addition, independent Claims 54, 56 and 58 were rejected under § 103(a) over Sato in view of U.S. Patent Application Publication 2001/0025312 (Obata) and further in view of Hirai.

All rejections are respectfully traversed, as detailed more fully below.

The invention generally concerns management of a multifunction network device on a network, wherein each multifunction network device includes plural function modules including a function module for controlling a printing capability and a function module for controlling a scanning capability. In response to reconfiguration events, the function modules may be deleted or deleted function modules may be retrieved. As a consequence, the functional capability of the multifunction network device can be changed by swapping function modules in and out of the device, such as in response to increases or decreases in demand for hardware resources of the multifunction network device.

In entering its rejection, the USPTO continues to take the position that there is no technological difference, in principle, between a local bus system within a single computer (such as that disclosed in Hirai) and a network system including plural multifunction network devices. Applicant continues to believe that such a position is technologically naive. In fact, there are significant differences between a local bus architecture within a single computer and network-based architecture, such that those of ordinary skill in the art would not consider it obvious to combine the two.

In support of his position, Applicant provides documentary evidence to prove that those of ordinary skill would not have considered it obvious to apply technology from a local bus system within a single computer to a distributed network system including plural multifunction network devices:

> G. Coulouris, et al. Distributed Systems: Concepts and Design (4th Edition), Addison Wesley; May 20, 2005 (hereinafter the "Coulouris Book"); and

Steve Muir, "The Seven Deadly Sins of Distributed Systems", Proceedings of the First Workshop on Real, Large Distributed Systems, December 5, 2004 (hereinafter the "Muir Article").

A copy of the entirety of the Muir Article is provided herewith. An excerpt from the Coulouris Book is also provided, comprising the title/copyright page, the table of contents, and the entirety of Chapter 1.

The Coulouris Book describes the general characteristics of a distributed network system. See Coulouris at page 2. According to Coulouris, the three significant characteristics of a distributed network system include concurrent execution, lack of a global clock, and independency of failures. These characteristics are different from those found within a self-contained single computer. These characteristics also lead the Coulouris Book to describe the challenges facing those of ordinary skill, when implementing a distributed network system, as compared to implementing a system on a self-contained single computer. See Coulouris at pages 16 to 25. According to Coulouris, these challenges include the following:

1. Heterogeneity: the variety and differences in networks, operating systems, computer hardware, programming languages and implementations.

2. Openness: the possible extensibility of systems by publishing specifications and documentations for key software interfaces.

3. Security: the confidentiality, integrity and availability against interference by malicious intruders.

4. Scalability: the ability of the system to remain effective even if there is a significant increase or decrease in the number of resources and users.

5. Failure handling: the ability to handle failures that are partial, meaning that some components of the distributed system might fail while others continue to execute.

6. Concurrency: the ability to allow requests from several clients to be processed such that they are all in a state of concurrent execution at the same time.

7. Transparency: concealment from the user and the application program of the separation of components in the distributed system, such that the system is perceived as a whole rather than a collection of independent components.

Thus, as described in the Coulouris Book, distributed network systems are far different from a local bus system in a single computer, as described in Hirai, and come with challenges so significant that those of ordinary skill in the art would not consider it obvious to transfer such technology to a distributed network system.

These point are echoed in the Muir Article, which describes "seven deadly sins" of distributed systems. According to the Muir Article, these seven deadly sins are:

1. Networks are unreliable in the worst possible way.
2. DNS does not make for a good naming system.
3. Local clocks are inaccurate and unreliable.
4. Large-scale systems always have inconsistencies.
5. Improbable events occur frequently in large systems.
6. Overutilization is the steady-state condition.
7. Limited system transparency hampers debugging.

Thus, as echoed by the Muir Article, those of ordinary skill in the art would not consider it obvious to combine technology from a local bus system within a single

computer (such as that disclosed in Hirai) into a distributed network system that includes plural multifunction network devices.

Moreover, with respect to specific language set out in the claims, all claims refer to a function module for a printing capability and a function module for a scanning capability, deletion of such function modules as appropriate in a situation where there is an increase of demand for hardware resources, and retrieval of deleted function modules as appropriate in a situation where there is a decrease in demand for the hardware resources. In a situation where a function module has been deleted, the multifunction device operates itself without the corresponding scanning capability or without the corresponding printing capability. In the case of retrieval of deleted functional modules, the multifunction device operates itself with scanning or printing capabilities, corresponding to the retrieved function module.

The art of record is not seen to disclose or to suggest anything similar to this notion.

It is therefore respectfully submitted that the claims herein recite subject matter that would not have been obvious from any of the art applied against it, alone or in any permissible combination, and allowance of the claims is respectfully requested.

Applicant's undersigned attorney may be reached in our Costa Mesa, California office at (714) 540-8700. All correspondence should continue to be directed to our below-listed address.

Respectfully submitted,

Attorney for Applicant
Michael K. O'Neill
Registration No.: 32,622

FITZPATRICK, CELLA, HARPER & SCINTO
30 Rockefeller Plaza
New York, New York 10112-3800
Facsimile: (212) 218-2200

FCHS_WS 2880265v1

DISTRIBUTED SYSTEMS

CONCEPTS AND DESIGN

Fourth edition

GEORGE COULOURIS
JEAN DOLLIMORE
TIM KINDBERG

Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoned.co.uk

First published 1988
Second edition 1994
Third edition 2001
Fourth edition 2005

ISBN 0 321 26354 5

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data
Coulouris, George F.
Distributed systems : concepts and design / George Coulouris, Jean Dollimore, Tim Kindberg.—4th ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-26354-5
1. Electronic data processing—Distributed processing. I. Dollimore, Jean. II. Kindberg, Tim. III. Title

QA76.9.D5C68 2005
004'.36—dc22

2005043613

1st Impression 2005

Typeset by the authors using FrameMaker
Printed and bound in the United States of America

The publisher's policy is to use paper manufactured from sustainable forests.

[redacted]

CONTENTS

1

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages. This definition leads to the following characteristics of distributed systems: concurrency of components, lack of a global clock and independent failures of components.

We give three examples of distributed systems:

- the Internet;
- an intranet, which is a portion of the Internet managed by an organization;
- mobile and ubiquitous computing.

The sharing of resources is a main motivation for constructing distributed systems. Resources may be managed by servers and accessed by clients or they may be encapsulated as objects and accessed by other client objects. The Web is discussed as an example of resource sharing and its main features are introduced.

The challenges arising from the construction of distributed systems are the heterogeneity of its components, openness, which allows components to be added or replaced, security, scalability – the ability to work well when the number of users increases – failure handling, concurrency of components and transparency.

1.1 Introduction

Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks, all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading *distributed systems*. In this book we aim to explain the characteristics of networked computers that impact system designers and implementors and to present the main concepts and techniques that have been developed to help in the tasks of designing and implementing systems that are based on them.

We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building or the same room. Our definition of distributed systems has the following significant consequences:

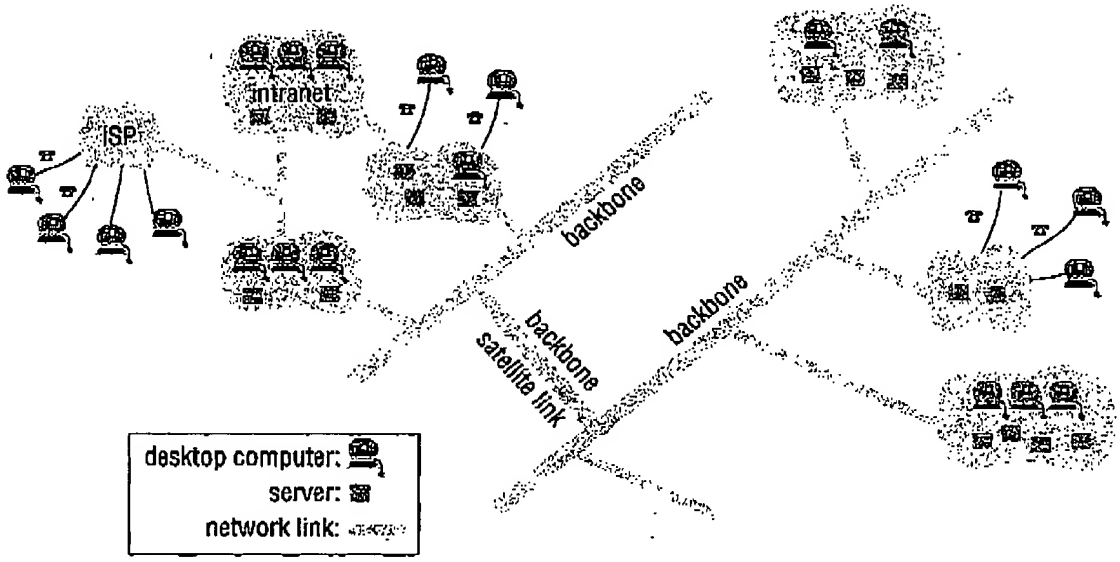
Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network. We will describe ways in which this extra capacity can be usefully deployed at many points in this book. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

No global clock: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network. Examples of these timing problems and solutions to them will be described in Chapter 11.

Independent failures: All computer systems can fail and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a *crash*) is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running. The consequences of this characteristic of distributed systems will be a recurring theme throughout the book.

The motivation for constructing and using distributed systems stems from a desire to share resources. The term 'resource' is a rather abstract one, but it best characterizes the range of things that can usefully be shared in a networked computer system. It extends

Figure 1.1 A typical portion of the Internet



from hardware components such as disks and printers to software-defined entities such as files, databases and data objects of all kinds. It includes the stream of video frames that emerges from a digital video camera and the audio connection that a mobile phone call represents.

The purpose of this chapter is to convey a clear view of the nature of distributed systems and the challenges that must be addressed in order to ensure that they are successful. Section 1.2 gives some key examples of distributed systems, the components from which they are constructed and their purposes. Section 1.3 explores the design of resource-sharing systems in the context of the World Wide Web. Section 1.4 describes the key challenges faced by the designers of distributed systems: heterogeneity, openness, security, scalability, failure handling, concurrency and the need for transparency.

1.2 Examples of distributed systems

Our examples are based on familiar and widely used computer networks: the Internet, intranets and the emerging technology of networks based on mobile devices. They are designed to exemplify the wide range of services and applications that are supported by computer networks and to begin the discussion of the technical issues that underlie their implementation.

1.2.1 The Internet

The Internet is a vast interconnected collection of computer networks of many different types. Figure 1.1 illustrates a typical portion of the Internet. Programs running on the

computers connected to it interact by passing messages, employing a common means of communication. The design and construction of the Internet communication mechanisms (the Internet protocols) is a major technical achievement, enabling a program running anywhere to address messages to programs anywhere else.

The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email and file transfer. (Indeed, the Web is sometimes incorrectly equated with the Internet). The set of services is open-ended – it can be extended by the addition of server computers and new types of service. The figure shows a collection of intranets – subnetworks operated by companies and other organizations. Internet Service Providers (ISPs) are companies that provide modem links and other types of connection to individual users and small organizations, enabling them to access services anywhere in the Internet as well as providing local services such as email and web hosting. The intranets are linked together by backbones. A backbone is a network link with a high transmission capacity, employing satellite connections, fibre optic cables and other high-bandwidth circuits.

Multimedia services are available in the Internet, enabling users to access audio and video data including music, radio and TV channels and to hold phone and video conferences. The capacity of the Internet to handle the special communication requirements of multimedia data is currently quite limited because it does not provide the necessary facilities to reserve network capacity for individual streams of data. Chapter 17 discusses the needs of distributed multimedia systems.

The implementation of the Internet and the services that it supports has entailed the development of practical solutions to many distributed system issues (including most of those defined in Section 1.4). We shall highlight those solutions throughout the book, pointing out their scope and their limitations where appropriate.

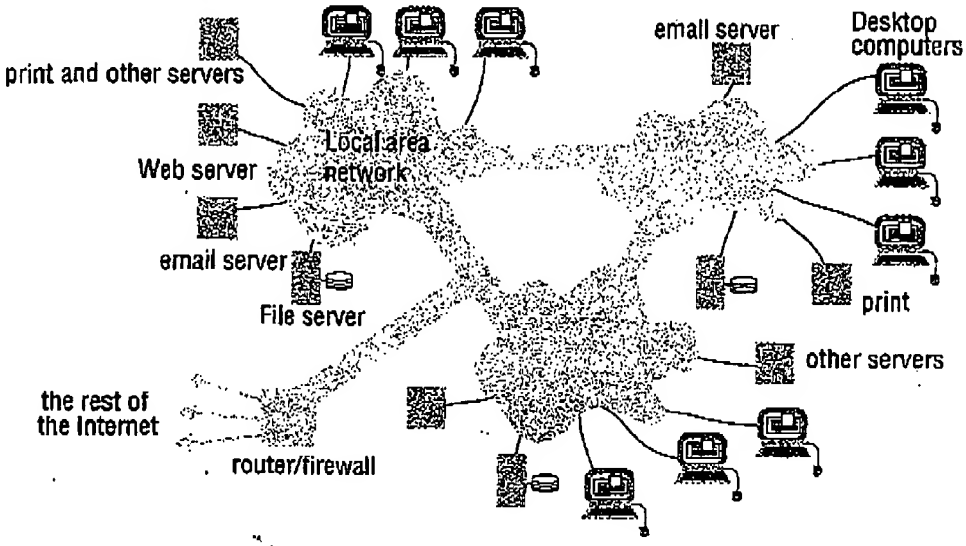
1.2.2 Intranets

An intranet is a portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies. Figure 1.2 shows a typical intranet. It is composed of several local area networks (LANs) linked by backbone connections. The network configuration of a particular intranet is the responsibility of the organization that administers it and may vary widely – ranging from a LAN on a single site to a connected set of LANs belonging to branches of a company or other organization in different countries.

An intranet is connected to the Internet via a router, which allows the users inside the intranet to make use of services elsewhere such as the Web or email. It also allows the users in other intranets to access the services it provides. Many organizations need to protect their own services from unauthorized use by possibly malicious users elsewhere. For example, a company will not want secure information to be accessible to users in competing organizations, and a hospital will not want sensitive patient data to be revealed. Companies also want to protect themselves from harmful programs such as viruses entering and attacking the computers in the intranet and possibly destroying valuable data.

The role of a *firewall* is to protect an intranet by preventing unauthorized messages leaving or entering. A firewall is implemented by filtering incoming and outgoing messages, for example according to their source or destination. A firewall

Figure 1.2 A typical intranet



might for example allow only those messages related to email and web access to pass into or out of the intranet that it protects.

Some organizations do not wish to connect their internal networks to the Internet at all. For example, police and other security and law enforcement agencies are likely to have at least some internal networks that are isolated from the outside world. Some military organizations disconnect their internal networks from the Internet at times of war. But even those organizations will wish to benefit from the huge range of application and system software that employs Internet communication protocols. The solution that is usually adopted by such organizations is to operate an intranet as described above, but without the connections to the Internet. Such an intranet can dispense with the firewall; or, to put it another way, it has the most effective firewall possible – the absence of any physical connections to the Internet.

The main issues arising in the design of components for use in intranets are:

- File services are needed to enable users to share data; the design of these is discussed in Chapter 8.
- Firewalls tend to impede legitimate access to services – when resource sharing between internal and external users is required, firewalls must be complemented by the use of fine-grained security mechanisms; these are discussed in Chapter 7.
- The cost of software installation and support is an important issue. These costs can be reduced by the use of system architectures such as network computers and thin clients, described in Chapter 2.

1.2.3 Mobile and ubiquitous computing

Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- Laptop computers.
- Handheld devices, including personal digital assistants (PDAs), mobile phones, pagers, video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing (also called *nomadic computing* [Kleinrock 1997]) is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their 'home' intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them. They can continue to access the Internet; they can continue to access resources in their home intranet; and there is increasing provision for users to utilize resources such as printers that are conveniently nearby as they move around. The latter is also known as *location-aware* or *context-aware computing*.

Ubiquitous computing [Weiser 1993] is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings. The term 'ubiquitous' is intended to suggest that small computing devices will eventually become so pervasive in everyday objects that they are scarcely noticed. That is, their computational behaviour will be transparently and intimately tied up with their physical function.

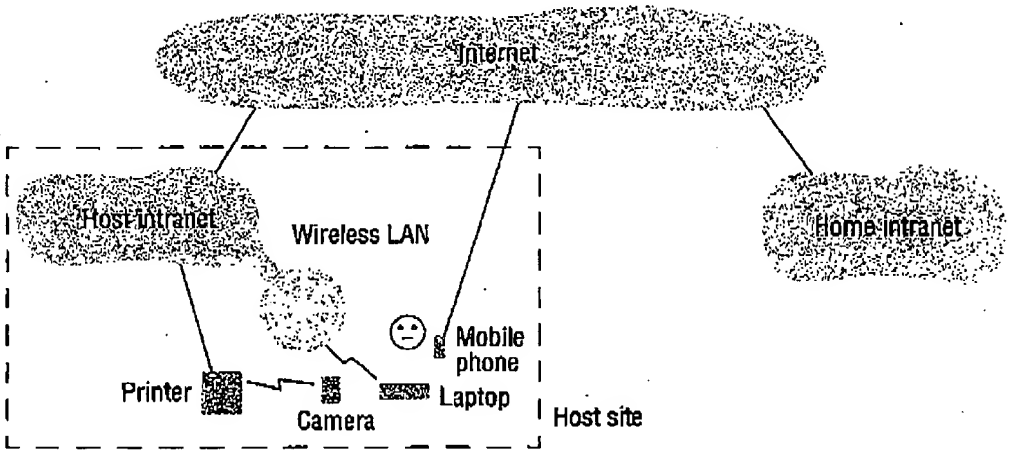
The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it would be convenient for users to control their washing machine and their hi-fi system from a 'universal remote control' device in the home. Equally, the washing machine could page the user via a smart badge or watch when the washing is done.

Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. But they are distinct, in general. Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital. Similarly, mobile computing has advantages even if it involves only conventional, discrete computers and devices such as laptops and printers.

Figure 1.3 shows a user who is visiting a host organization. The figure shows the user's home intranet and the host intranet at the site that the user is visiting. Both intranets are connected to the rest of the Internet.

The user has access to three forms of wireless connection. Their laptop has a means of connecting to the host's wireless LAN. This network provides coverage of a few hundreds of metres (a floor of a building, say). It connects to the rest of the host intranet via a gateway. The user also has a mobile (cellular) telephone, which is connected to the Internet. The phone gives access to pages of simple information, which

Figure 1.3 Portable and handheld devices in a distributed system



it presents on its small display. Finally, the user carries a digital camera, which can communicate over a personal area wireless network (with range up to about 10m) with a device such as a printer.

With a suitable system infrastructure, the user can perform some simple tasks in the host site using the devices they carry. While journeying to the host site, the user can fetch the latest stock prices from a web server using the mobile phone. During the meeting with their hosts, the user can show them a recent photograph by sending it from the digital camera directly to a suitably enabled printer in the meeting room. This requires only the wireless link between the camera and printer. And they can in principle send a document from their laptop to the same printer, utilizing the wireless LAN and wired Ethernet links to the printer.

Mobile and ubiquitous computing are a lively area of research and they are the subject of Chapter 16.

1.3 Resource sharing and the Web

Users are so accustomed to the benefits of resource sharing that they may easily overlook their significance. We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.

Looked at from the point of view of hardware provision, we share equipment such as printers and disks to reduce costs. But of far greater significance to users is their sharing of the higher-level resources that play a part in their applications and in their everyday work and social activities. For example, users are concerned with sharing data in the form of a shared database or a set of web pages – not the disks and processors that those are implemented on. Similarly, users think in terms of shared resources such as a search engine or a currency converter, without regard for the server or servers that provide these.

In practice, patterns of resource sharing vary widely in their scope and in how closely users work together. At one extreme, a search engine on the Web provides a facility to users throughout the world, users who need never come into contact with one another directly. At the other extreme, in *computer-supported cooperative working* (CSCW), a group of users who cooperate directly share resources such as documents in a small, closed group. The pattern of sharing and the geographic distribution of particular users determines what mechanisms the system must supply to coordinate users' actions.

We use the term *service* for a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications. For example, we access shared files through a file service; we send documents to printers through a printing service; we buy goods through an electronic payment service. The only access we have to the service is via the set of operations that it exports. For example, a file service provides *read*, *write* and *delete* operations on files.

The fact that services restrict resource access to a well-defined set of operations is in part standard software engineering practice. But it also reflects the physical organization of distributed systems. Resources in a distributed system are physically encapsulated within computers and can only be accessed from other computers by communication. For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.

The term *server* is probably familiar to most readers. It refers to a running program (a *process*) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately. The requesting processes are referred to as *clients*. Requests are sent in messages from clients to a server and replies are sent in messages from the server to the clients. When the client sends a request for an operation to be carried out, we say that the client *invokes an operation* upon the server. A complete interaction between a client and a server, from the point when the client sends its request to when it receives the server's response, is called a *remote invocation*.

The same process may be both a client and a server, since servers sometimes invoke operations on other servers. The terms 'client' and 'server' apply only to the roles played in a single request. In so far as they are distinct, clients are active and servers are passive; servers run continuously, whereas clients last only as long as the applications of which they form a part.

Note that by default the terms 'client' and 'server' refer to *processes* rather than the computers that they execute upon, although in everyday parlance those terms also refer to the computers themselves. Another distinction, which we shall discuss in Chapter 5, is that in a distributed system written in an object-oriented language, resources may be encapsulated as objects and accessed by client objects, in which case we speak of a *client object* invoking a method upon a *server object*.

Many, but certainly not all, distributed systems can be constructed entirely in the form of interacting clients and servers. The World Wide Web, email and networked printers all fit this model. We discuss alternatives to client-server systems in Chapter 2.

An executing web browser is an example of a client. The web browser communicates with a web server, to request web pages from it. We now examine the Web in more detail.

1.3.1 The World Wide Web

The World Wide Web [www.w3.org I, Berners-Lee 1991] is an evolving system for publishing and accessing resources and services across the Internet. Through commonly available web browsers, users retrieve and view documents of many types, listen to audio streams and view video streams, and interact with an unlimited set of services.

The Web began life at the European centre for nuclear research (CERN), Switzerland, in 1989 as a vehicle for exchanging documents between a community of physicists connected by the Internet [Berners-Lee 1999]. A key feature of the Web is that it provides a *hypertext* structure among the documents that it stores, reflecting the users' requirement to organize their knowledge. This means that documents contain *links* (or *hyperlinks*) – references to other documents and resources that are also stored in the Web.

It is fundamental to the user's experience of the Web that when he or she encounters a given image or piece of text within a document, this will frequently be accompanied by links to related documents and other resources. The structure of links can be arbitrarily complex and the set of resources that can be added is unlimited – the 'web' of links is indeed world-wide. Bush [1945] conceived of hypertextual structures over fifty years ago; it was with the development of the Internet that this idea could be manifested on a world-wide scale.

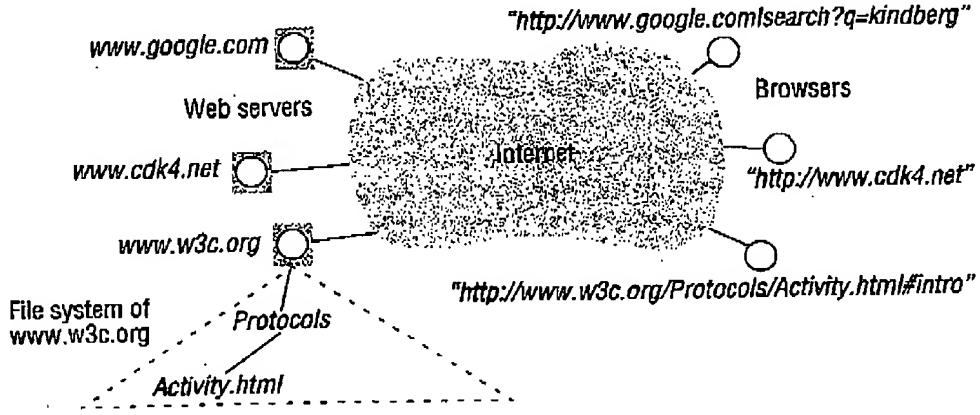
The Web is an *open* system: it can be extended and implemented in new ways without disturbing its existing functionality (see Section 1.4.2). First, its operation is based on communication standards and document standards that are freely published and widely implemented. For example, there are many types of browser, each in many cases implemented on several platforms; and there are many implementations of web servers. Any conformant browser can retrieve resources from any conformant server. So users have access to browsers on the majority of the devices that they use, from mobile phones to desktop computers.

Second, the Web is open with respect to the types of resource that can be published and shared on it. At its simplest, a resource on the Web is a web page or some other type of *content* that can be stored in a file and presented to the user, such as program files, media files, and documents in PostScript or Portable Document Format. If somebody invents, say, a new image-storage format, then images in this format can immediately be published on the Web. Users require a means of viewing images in this new format, but browsers are designed to accommodate new content-presentation functionality in the form of 'helper' applications and 'plug-ins'.

The Web has moved beyond these simple data resources to encompass services, such as electronic purchasing of goods. It has evolved without changing its basic architecture. The Web is based on three main standard technological components:

- The HyperText Markup Language (HTML) is a language for specifying the contents and layout of pages as they are displayed by web browsers.
- Uniform Resource Locators (URLs), which identify documents and other resources stored as part of the Web. Chapter 9 discusses other terms for web identifiers.
- A client-server system architecture, with standard rules for interaction (the HyperText Transfer Protocol – HTTP) by which browsers and other clients fetch documents and other resources from web servers. Figure 1.4 shows some web

Figure 1.4 Web servers and web browsers



servers, and browsers making requests of them. It is an important feature that users may locate and manage their own web servers anywhere on the Internet.

We now discuss these components in turn, and in so doing explain the operation of browsers and web servers when a user fetches web pages and clicks on the links within them.

HTML ◊ The HyperText Markup Language [www.w3.org II] is used to specify the text and images that make up the contents of a web page, and to specify how they are laid out and formatted for presentation to the user. A web page contains such structured items as headings, paragraphs, tables and images. HTML is also used to specify links and which resources are associated with them.

Users either produce HTML by hand, using a standard text editor, or they can use an HTML-aware 'wysiwyg' editor that generates HTML from a layout that they create graphically. A typical piece of HTML text follows:

```
<IMG SRC = "http://www.cdk4.net/WebExample/Images/earth.jpg" >          1
                                                                        2
<P>
Welcome to Earth! Visitors may also be interested in taking a look at the   3
<A HREF = "http://www.cdk4.net/WebExample/moon.html" >Moon</A>.          4
</P>                                                                    5
```

This HTML text is stored in a file that a web server can access – let us say the file *earth.html*. A browser retrieves the contents of this file from a web server – in this case a server on a computer called *www.cdk4.net*. The browser reads the content returned by the server and renders it into formatted text and images laid out on a web page in the familiar fashion. Only the browser – not the server – interprets the HTML text. But the server does inform the browser of the type of content it is returning, to distinguish it from, say, a document in PostScript. The server can infer the content type from the filename extension '.html'.

Note that the HTML directives, known as *tags*, are enclosed by angle brackets, such as *<P>*. Line 1 of the example identifies a file containing an image for

presentation. Its URL is *http://www.cdk4.net/WebExample/Images/earth.jpg*. Lines 2 and 5 are directives to begin and end a paragraph, respectively. Lines 3 and 4 contain text to be displayed on the web page in the standard paragraph format.

Line 4 specifies a link in the web page. It contains the word 'Moon' surrounded by two related HTML tags *<A HREF...>* and *</A>*. The text between these tags is what appears in the link as it is presented on the web page. Most browsers are configured to show the text of links underlined by default, so what the user will see in that paragraph is:

Welcome to Earth! Visitors may also be interested in taking a look at the Moon.

The browser records the association between the link's displayed text and the URL contained in the *<A HREF...>* tag – in this case:

http://www.cdk4.net/WebExample/moon.html

When the user clicks on the text, the browser retrieves the resource identified by the corresponding URL and presents it to the user. In the example, the resource is an HTML file specifying a web page about the Moon.

URLs ◊ The purpose of a Uniform Resource Locator [www.w3.org III] is to identify a resource. Indeed, the term used in web architecture documents is Uniform Resource *Identifier* (URI) but in this book the better-known term URL will be used when no confusion can arise. Browsers examine URLs in order to access the corresponding resources. Sometimes the user types a URL into the browser. More commonly, the browser looks up the corresponding URL when the user clicks on a link or selects one of their 'bookmarks'; or when the browser fetches a resource embedded in a web page, such an an image.

Every URL, in its full, absolute form, has two top-level components:

scheme : scheme-specific-identifier

The first component, the 'scheme', declares which type of URL this is. URLs are required to identify a variety of resources. For example, *mailto:joe@anISP.net* identifies a user's email address; *ftp://ftp.downloadIt.com/software/aProg.exe* identifies a file that is to be retrieved using the File Transfer Protocol (FTP) rather than the more commonly used protocol HTTP. Other examples of schemes are 'nntp' (used to specify a Usenet news group), and 'mid' (used to identify an email message).

The Web is open with respect to the types of resources it can be used to access, by virtue of the scheme designators in URLs. If somebody invents a useful new type of 'widget' resource – perhaps with its own addressing scheme for locating widgets and its own protocol for accessing them – then the world can start using URLs of the form *widget:...* . Of course, browsers must be given the capability to use the new 'widget' protocol, but this can be done by adding a plug-in.

HTTP URLs are the most widely used, for accessing resources using the standard HTTP protocol. An HTTP URL has two main jobs to do: to identify which web server maintains the resource, and to identify which of the resources at that server is required. Figure 1.4 shows three browsers issuing requests for resources managed by three web servers. The topmost browser is issuing a query to a search engine. The middle browser requires the default page of another web site. The bottommost browser requires a web page that is specified in full, including a path name relative to the server. The files for a

given web server are maintained in one or more sub-trees (directories) of the server's file system, and each resource is identified by a path name relative to the server.

In general, HTTP URLs are of the following form:

http:// servername [:port] [/pathName] [?query][#fragment]

– where items in square brackets are optional. A full HTTP URL always begins with the string 'http://' followed by a server name, expressed as a Domain Name System (DNS) name (see Section 9.2). The server's DNS name is optionally followed by the number of the 'port' on which the server listens for requests (see Chapter 4) – which is 80 by default. Then comes an optional path name of the server's resource. If this is absent then the server's default web page is required. Finally, the URL optionally ends in a query component – for example, when a user submits the entries in a form such as a search engine's query page – and/or a fragment identifier, which identifies a component of the resource.

Consider the URLs:

http://www.cdk4.net
http://www.w3.org/Protocols/Activity.html#intro
http://www.google.com/search?q=kindberg

These can be broken down as follows:

DNS name	pathName	query	fragment
www.cdk4.net	(default)	(none)	(none)
www.w3.org	Protocols/Activity.html	(none)	intro
www.google.com	search	q=kindberg	(none)

The first URL designates the default page supplied by *www.cdk4.net*. The next identifies a fragment of an HTML file whose path name is *Protocols/Activity.html* relative to the server *www.w3.org*. The fragment's identifier (specified after the '#' character in the URL) is *intro*, and a browser will search for that fragment identifier within the HTML text after it has downloaded the whole file. The third URL specifies a query to a search engine. The path identifies a program called 'search', and the string after the '?' character encodes a query string supplied as arguments to this program. We discuss URLs that identify programmatic resources in more detail when we consider more advanced features below.

Publishing a resource: While the Web has a clearly defined model for accessing a resource from its URL, the exact methods for publishing resources on the Web are dependent upon the web server implementation. The simplest method of publishing a resource on the Web is to place the corresponding file in a directory that the web server can access. Knowing the name of the server *S* and a path name for the file *P* that the server can recognize, the user then constructs the URL as *http://S/P*. The user puts this URL in a link from an existing document or distributes the URL to other users, for example by email.

There are certain path name conventions that servers recognize. For example, a path name beginning *~joe* is by convention in a subdirectory *public_html* of user joe's

home directory. Similarly, a path name that ends in a directory name rather than a simple file conventionally refers to a file in that directory called *index.html*.

Huang *et al.* [2000] provide a model for inserting content into the Web with minimal human intervention. This is particularly relevant where users need to extract content from a variety of devices, such as cameras, for publication in web pages.

HTTP ◊ The HyperText Transfer Protocol [www.w3.org IV] defines the ways in which browsers and other types of client interact with web servers. Chapter 4 will consider HTTP in more detail, but here we outline its main features (restricting our discussion to the retrieval of resources in files):

Request-reply interactions: HTTP is a 'request-reply' protocol. The client sends a request message to the server containing the URL of the required resource. The server looks up the path name and, if it exists, sends back the file's contents in a reply message to the client. Otherwise, it sends back an error response such as the familiar '404 Not Found'.

Content types: Browsers are not necessarily capable of handling every type of content. When a browser makes a request, it includes a list of the types of content it prefers – for example, in principle it may be able to display images in 'GIF' format but not 'JPEG' format. The server may be able to take this into account when it returns content to the browser. The server includes the content type in the reply message so that the browser will know how to process it. The strings that denote the type of content are called MIME types, and they are standardized in RFC 1521 [Freed and Borenstein 1996]. For example, if the content is of type 'text/html' then a browser will interpret the text as HTML and display it; if the content is of type 'image/GIF' then the browser will render it as an image in 'GIF' format; if the content type is 'application/zip' then it is data compressed in 'zip' format, and the browser will launch an external helper application to decompress it. The set of actions that a browser will take for a given type of content is configurable, and readers may care to check these settings for their own browsers.

One resource per request: Clients specify one resource per HTTP request. If a web page contains nine images, say, then the browser will issue a total of ten separate requests to obtain the entire contents of the page. Browsers typically make several requests concurrently, to reduce the overall delay to the user.

Simple access control: By default, any user with network connectivity to a web server can access any of its published resources. If users wish to restrict access to a resource, then they can configure the server to issue a 'challenge' to any client that requests it. The corresponding user then has to prove that they have the right to access the resource, for example, by typing in a password.

Dynamic pages ◊ So far we have described how users can publish web pages and other content stored in files on the Web. However, much of the users' experience of the Web is that of interacting with services rather than retrieving data. For example, when purchasing an item at an online store, the user often fills out a *web form* to give their personal details or to specify exactly what they wish to purchase. A web form is a web page containing instructions for the user and input widgets such as text fields and check boxes. When the user submits the form (usually by pressing a button or the 'return' key),

the browser sends an HTTP request to a web server, containing the values that the user has entered.

Since the result of the request depends upon the user's input, the server has to *process* the user's input. Therefore, the URL or its initial component designates a *program* on the server, not a file. If the user's input is reasonably short then it is usually sent as the *query* component of the URL, following a '?' character (otherwise it is sent as additional data in the request). For example, a request containing the following URL invokes a program called 'search' at www.google.com and specifies a query string of 'kindberg': *http://www.google.com/search?q=kindberg*.

That 'search' program produces HTML text as its output, and the user will see a listing of pages that contain the word 'kindberg'. (The reader may care to enter a query into their favourite search engine and notice the URL that the browser displays when the result is returned.) The server returns the HTML text that the program generates just as though it had retrieved it from a file. In other words, the difference between static content fetched from a file and content that is dynamically generated is transparent to the browser.

A program that web servers run to generate content for their clients is often referred to as a Common Gateway Interface (CGI) program. A CGI program may have any application-specific functionality, as long as it can parse the arguments that the client provides to it and produce content of the required type (usually HTML text). The program will often consult or update a database in processing the request.

Downloaded code: A CGI program runs at the server. Sometimes the designers of web services require some service-related code to run inside the browser, at the user's computer. For example, code written in Javascript [www.netscape.com] is often downloaded with a web form in order to provide better-quality interaction with the user than that supported by HTML's standard widgets. A Javascript-enhanced page can give the user immediate feedback on invalid entries (instead of forcing the user to check the values at the server, which would take much longer). Javascript can also be used to update parts of a web page's contents without fetching an entire new version of the page and re-rendering it.

Javascript has quite limited functionality. By contrast, an *applet* is an application that the browser automatically downloads and runs when it fetches a corresponding web page. Applets may access the network and provide customized user interfaces, using the facilities of the Java language [java.sun.com, Flanagan 2002]. For example, 'chat' applications are sometimes implemented as applets that run on the users' browsers, together with a server program. The applets send the users' text to the server, which in turn distributes it to all the applets for presentation to the user. We discuss applets in more detail in Section 2.2.3.

Web Services ◊ So far we have discussed the Web largely from the point of view of a user operating a browser. But programs other than browsers can be clients of the Web, too; indeed, programmatic access to web resources is commonplace.

However, by themselves, the HTML and HTTP standards are somewhat lacking for programmatic interoperation. First, there is an increasing need to exchange many types of structured data on the Web, but HTML is limited in that it is not extensible to applications beyond information browsing. HTML has a static set of structures such as paragraphs, and they are bound up with the way that the data is to be presented to users.

The Extensible Markup Language (XML) (see Section 4.3.3) has been designed as a way of representing data in standard, structured, application-specific forms. For example, XML is used to describe the capabilities of devices and to describe personal information held about users. XML is a meta-language for describing data, which makes data portable between applications.

Second, HTTP provides no structure for specifying the service-specific operations that can be invoked on web resources, or the operations' arguments and error responses For example, in the store at amazon.com, web service operations include one to order a book and another to check the current state of an order. The flip-side of that flexibility can be a lack of robustness in how software operates. Chapter 19 takes an in-depth look at the web services framework, which enables the designers of web services to specify to programmers exactly how clients must access them.

Discussion of the Web ◊ The Web's phenomenal success rests upon the relative ease with which many individual and organizational sources can publish resources, the suitability of its hypertext structure for organizing many types of information, and the openness of its system architecture. The standards upon which its architecture is based are simple and they were widely published at an early stage. They have enabled many new types of resource and service to be integrated.

The Web's success belies some design problems. First, its hypertext model is lacking in some respects. If a resource is deleted or moved, then so-called 'dangling' links to that resource may still remain, causing frustration for users. And there is the familiar problem of users getting 'lost in hyperspace'. Users often find themselves confused, following many disparate links, referencing pages from a disparate collection of sources, and of dubious reliability in some cases.

Search engines are an alternative to following links as a means of finding information on the Web, but these are notoriously imperfect at producing what the user specifically intends. One approach to this problem, exemplified in the Resource Description Framework [www.w3.org V], is to produce standard vocabularies, syntax and semantics for expressing metadata about the things in our world, and to encapsulate that metadata in corresponding web resources for programmatic access. Rather than searching for words that occur in web pages, programs can then, in principle, perform searches against the metadata to compile lists of related links based on semantic matching. Collectively, the web of linked metadata resources is what is meant by the *semantic web*.

As a system architecture the Web faces problems of scale. Popular web servers may experience many 'hits' per second, and as a result the response to users can be slow. Chapter 2 describes the use of caching in browsers and proxy servers to increase responsiveness, and the division of the server's load across clusters of computers. But the Web's client-server architecture means that it has no efficient means of keeping users up to date with the latest versions of pages. Users have to press their browser's 'reload' button to ensure that they have the latest information, and browsers are forced to communicate with servers to check whether a local copy of a resource is still valid.

Finally, a web page is not always a satisfactory user interface. The interface widgets defined for HTML are limited, and designers often include applets or many images in web pages to make them look and function more acceptably. There is a consequent increase in the download time.

1.4 Challenges

The examples in Section 1.2 are intended to illustrate the scope of distributed systems and to suggest the issues that arise in their design. Although distributed systems are to be found everywhere, their design is quite simple and there is still a lot of scope to develop more ambitious services and applications. Many of the challenges discussed in this section have already been met, but future designers need to be aware of them and to be careful to take them into account.

1.4.1 Heterogeneity

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers.

Although the Internet consists of many different sorts of network (illustrated in Figure 1.1), their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network. Chapter 3 explains how the Internet protocols are implemented over a variety of different networks.

Data types such as integers may be represented in different ways on different sorts of hardware for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware.

Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another.

Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

Middleware ◊ The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying

networks, hardware, operating systems and programming languages. The Common Object Request Broker (CORBA), which is described in Chapters 4, 5 and 20, is an example. Some middleware, such as Java Remote Method Invocation (RMI) (see Chapter 5) supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the difference of the underlying networks. But all middleware deals with the differences in operating systems and hardware – how this is done is the main topic of Chapter 4.

In addition to solving the problems of heterogeneity, middleware provides a uniform computational model for use by the programmers of servers and distributed applications. Possible models include remote object invocation, remote event notification, remote SQL access and distributed transaction processing. For example, CORBA provides remote object invocation, which allows an object in a program running on one computer to invoke a method of an object in a program running on another computer. Its implementation hides the fact that messages are passed over a network in order to send the invocation request and its reply.

Heterogeneity and mobile code ◊ The term *mobile code* is used to refer to code that can be sent from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system. For example, executable files sent as email attachments by Windows/x86 users will not run on an x86 computer running Linux or a Macintosh computer running Mac OS X.

The *virtual machine* approach provides a way of making code executable on any hardware: the compiler for a particular language generates code for a virtual machine instead of a particular hardware order code for example, the Java compiler produces code for the Java virtual machine, which needs to be implemented once for each type of hardware to enable Java programs to run. However, the Java solution is not generally applicable to programs written in other languages.

1.4.2 Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures, which are usually cumbersome and slow-moving.

However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people.

The designers of the Internet protocols introduced a series of documents called 'Requests For Comments', or RFCs, each of which is known by a number. The specifications of the Internet communication protocols were published in this series in the early 1980s, followed by specifications for applications that run over them, such as file transfer, email and telnet by the mid-1980s. This practice has continued and forms the basis of the technical documentation of the Internet. This series includes discussions as well as the specifications of protocols. Copies can be obtained from [www.ietf.org]. Thus the publication of the original Internet communication protocols has enabled a variety of Internet systems and applications including the Web to be built. RFCs are not the only means of publication. For example, CORBA is published through a series of technical documents, including a complete specification of the interfaces of its services. See [www.omg.org].

Systems that are designed to support resource sharing in this way are termed *open distributed systems* to emphasize the fact that they are extensible. They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services and the re-implementation of old ones, enabling application programs to share resources. A further benefit that is often cited for open systems is their independence from individual vendors.

To summarize:

- Open systems are characterized by the fact that their key interfaces are published.
- Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
- Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

1.4.3 Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality (protection against disclosure to unauthorized individuals); integrity (protection against alteration or corruption); and availability (protection against interference with the means to access the resources).

Section 1.1 pointed out that although the Internet allows a program in one computer to communicate with a program in another computer irrespective of its location, security risks are associated with allowing free access to all of the resources in an intranet. Although a firewall can be used to form a barrier around an intranet, restricting the traffic that can enter and leave, this does not deal with ensuring the appropriate use of resources by users within an intranet, or with the appropriate use of resources in the Internet, that are not protected by firewalls.

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example:

1. A doctor might request access to hospital patient data or send additions to that data.

2. In electronic commerce and banking, users send their credit card numbers across the Internet.

In both examples, the challenge is to send sensitive information in a message over a network in a secure manner. But security is not just a matter of concealing the contents of messages – it also involves knowing for sure the identity of the user or other agent on whose behalf a message was sent. In the first example, the server needs to know that the user is really a doctor and in the second example, the user needs to be sure of the identity of the shop or bank with which they are dealing. The second challenge here is to identify a remote user or other agent correctly. Both of these challenges can be met by the use of encryption techniques developed for this purpose. They are used widely in the Internet and are discussed in Chapter 7.

However, the following two security challenges have not yet been fully met:

Denial of service attacks: Another security problem is that a user may wish to disrupt a service for some reason. This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it. This is called a *denial of service* attack. From time to time, there have been several recent denial of service attacks on well-known web services. Currently such attacks are countered by attempting to catch and punish the perpetrators after the event, but that is not a general solution to the problem. Counter-measures based on improvements in the management of networks are under development and these will be touched on in Chapter 3.

Security of mobile code: Mobile code needs to be handled with care. Consider someone who receives an executable program as an electronic mail attachment: the possible effects of running the program are unpredictable; for example, it may seem to display an interesting picture but in reality it may access local resources, or perhaps be part of a denial of service attack. Some measures for securing mobile code are outlined in Chapter 7.

1.4.4 Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The Internet provides an illustration of a distributed system in which the number of computers and services has increased dramatically. Figure 1.5 shows the increase in the number of computers in the Internet during the 24 years up to 2003, and Figure 1.6 shows the increasing number of computers and web servers during the ten-year history of the Web up to 2004, see [zakon.org].

The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computers increases. It must be possible to add server computers to avoid the performance bottleneck that would arise if a single file server had to

Figure 1.5 Computers (with registered IP addresses) in the Internet

Date	Computers	Web Servers
1979, Dec.	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866
2003, Jan.	171,638,297	35,424,956

handle all file access requests. In general, for a system with n users to be scalable, the quantity of physical resources required to support them should be at most $O(n)$ – that is, proportional to n. For example, if a single file server can support 20 users, then two such servers should be able to support 40 users. Although that sounds an obvious goal, it is not necessarily easy to achieve in practice, as we show in Chapter 8.

Controlling the performance loss: Consider the management of a set of data whose size is proportional to the number of users or resources in the system, for example the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System, which is used mainly to look up DNS names such as www.amazon.com. Algorithms that use hierarchic structures scale better than those that use linear structures. But even with hierarchic structures an increase in size will result in some loss in performance: the time taken to access hierarchically structured data is $O(\log n)$, where n is the size of the set of data. For a system to be scalable, the maximum performance loss should be no worse than this.

Figure 1.6 Computers vs. web servers in the Internet

Date	Computers	Web servers[1]	Percentage
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July		42,298,371	

[1] one web server may be hosted at multiple sites

Preventing software resources running out: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). In the late 1970s, it was decided to use 32 bits for this purpose, but as will be explained in Chapter 3 the supply of available Internet addresses is running out. For this reason, a new version of the protocol with 128-bit Internet addresses is being adopted and this will require modifications to many software components. To be fair to the early designers of the Internet, there is no correct solution to this problem. It is difficult to predict the demand that will be put on a system years ahead. Moreover, over-compensating for future growth may be worse than adapting to a change when we are forced to – larger Internet addresses will occupy extra space in messages and in computer storage.

Avoiding performance bottlenecks: In general, algorithms should be decentralized to avoid having performance bottlenecks. We illustrate this point with reference to the predecessor of the Domain Name System in which the name table was kept in a single master file that could be downloaded to any computers that needed it. That was fine when there were only a few hundred computers in the Internet, but it soon became a serious performance and administrative bottleneck. The Domain Name System removed this bottleneck by partitioning the name table between servers located throughout the Internet and administered locally – see Chapters 3 and 9.

Some shared resources are accessed very frequently; for example, many users may access the same Web page, causing a decline in performance. We shall see in Chapter 2 that caching and replication may be used to improve the performance of resources that are very heavily used.

Ideally, the system and application software should not need to change when the scale of the system increases, but this is difficult to achieve. The issue of scale is a dominant theme in the development of distributed systems. The techniques that have been successful are discussed extensively in this book. They include the use of replicated data (Chapter 15), the associated technique of caching (Chapters 2 and 8) and the deployment of multiple servers to handle commonly performed tasks, enabling several similar tasks to be performed concurrently.

1.4.5 Failure handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or they may stop before they have completed the intended computation. We shall discuss and classify a range of possible failure types that can occur in the processes and networks that comprise a distributed system in Chapter 2.

Failures in a distributed system are partial – that is, some components fail while others continue to function. Therefore the handling of failures is particularly difficult. The following techniques for dealing with failures are discussed throughout the book:

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file. Chapter 2 explains that is difficult or even impossible to detect some other failures such as a remote crashed server in the Internet. The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

Masking failures: Some failures that have been detected can be hidden or made less severe. Two examples of hiding failures:

1. messages can be retransmitted when they fail to arrive;
2. file data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Just dropping a message that is corrupted is an example of making a fault less severe – it could be retransmitted. The reader will probably realize that the techniques described for hiding failures are not guaranteed to work in the worst cases; for example, the data on the second disk may be corrupted too, or the message may not get through in a reasonable time however often it is retransmitted.

Tolerating failures: Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components. Their clients can be designed to tolerate failures, which generally involves the users tolerating them as well. For example, when a web browser cannot contact a web server, it does not make the user wait for ever while it keeps on trying – it informs the user about the problem, leaving them free to try again later. Services that tolerate failures are discussed in the paragraph on redundancy below.

Recovery from failures: Recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed. In general, the computations performed by some programs will be incomplete when a fault occurs, and the permanent data that they update (files and other material stored in permanent storage) may not be in a consistent state. Recovery is described in Chapter 14.

Redundancy: Services can be made to tolerate failures by the use of redundant components. Consider the following examples:

1. There should always be at least two different routes between any two routers in the Internet.
2. In the Domain Name System, every name table is replicated in at least two different servers.
3. A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server; the servers can be designed to detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

The design of effective techniques for keeping replicas of rapidly changing data up-to-date without excessive loss of performance is a challenge. Approaches are discussed in Chapter 15.

Distributed systems provide a high degree of availability in the face of hardware faults. The *availability* of a system is a measure of the proportion of time that it is available for use. When one of the components in a distributed system fails, only the work that was using the failed component is affected. A user may move to another computer if the one that they were using fails; a server process can be started on another computer.

1.4.6 Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time.

The process that manages a shared resource could take one client request at a time. But that approach limits throughput. Therefore services and applications generally allow multiple client requests to be processed concurrently. To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads. In this case it is possible that several threads may be executing concurrently within an object, in which case their operations on the object may conflict with one another and produce inconsistent results. For example, if two concurrent bids at an auction are 'Smith: $122' and 'Jones: $111', and the corresponding operations are interleaved without any control, then they might get stored as 'Smith: $111' and 'Jones: $122'.

The moral of this story is that any object that represents a shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment. This applies not only to servers but also to objects in applications. Therefore any programmer who takes an implementation of an object that was not intended for use in a distributed system must do whatever is necessary to make it safe in a concurrent environment.

For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems. This topic and its extension to collections of distributed shared objects are discussed in Chapters 6 and 13.

1.4.7 Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

The ANSA Reference Manual [ANSA 1989] and the International Organization for Standardization's Reference Model for Open Distributed Processing (RM-ODP) [ISO 1992] identify eight forms of transparency. We have paraphrased the original ANSA definitions, replacing their migration transparency with our own mobility transparency, whose scope is broader:

Access transparency enables local and remote resources to be accessed using identical operations.

Location transparency enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.

Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

The two most important transparencies are access and location transparency; their presence or absence most strongly affects the utilization of distributed resources. They are sometimes referred to together as *network transparency*.

As an illustration of access transparency, consider a graphical user interface with folders, which is the same whether the files inside the folder are local or remote. Another example is an API for files that uses the same operations to access both local and remote files (see Chapter 8). As an example of a lack of access transparency, consider a distributed system that does not allow you to access files on a remote computer unless you make use of the ftp program to do so.

Web resource names or URLs are location-transparent because the part of the URL that identifies a web server domain name refers to a computer name in a domain, rather than to an Internet address. However, URLs are not mobility-transparent, because someone's personal web page cannot move to their new place of work in a different domain – all of the links in other pages will still point to the original page.

In general, identifiers such as URLs that include the domain names of computers prevent replication transparency. Although the DNS allows a domain name to refer to several computers, it picks just one of them when it looks up a name. Since a replication scheme generally needs to be able to access all of the participating computers, it would need to access each of the DNS entries by name.

As an illustration of the presence of network transparency, consider the use of an electronic mail address such as *Fred.Flintstone@stoneit.com*. The address consists of a user's name and a domain name. Note that although mail programs accept user names for local users, they do append the local domain name. Sending mail to such a user does not involve knowing their physical or network location. Nor does the procedure to send a mail message depend upon the location of the recipient. Thus electronic mail within the Internet provides both location and access transparency (that is, network transparency).

Failure transparency can also be illustrated in the context of electronic mail, which is eventually delivered, even when servers or communication links fail. The faults are masked by attempting to retransmit messages until they are successfully delivered, even if it takes several days. Middleware generally converts the failures of networks and processes into programming-level exceptions (see Chapter 5 for an explanation).

To illustrate mobility transparency, consider the case of mobile phones. Suppose that both caller and callee are travelling by train in different parts of a country, moving from one environment (cell) to another. We regard the caller's phone as the client and the callee's phone as a resource. The two phone users making the call are unaware of the mobility of the phones (the client and the resource) between cells.

Transparency hides and renders anonymous the resources that are not of direct relevance to the task in hand from users and application programmers. For example, it is generally desirable for similar hardware resources to be allocated interchangeably to perform a task – the identity of a processor used to execute a process is generally hidden from the user and remains anonymous. As pointed out in Section 1.2.3, this may not always be what is required: for example, a traveller who attaches a laptop computer to the local network in each office visited should make use of local services such as the send mail service, using different servers at each location. Even within a building, it is normal to arrange for a document to be printed at a particular, named printer: usually one that is near to the user.

1.5 Summary

Distributed systems are everywhere. The Internet enables users throughout the world to access its services wherever they may be located. Each organization manages an intranet, which provides local services and Internet services for local users and generally provides services to other users in the Internet. Small distributed systems can be constructed from mobile computers and other small computational devices that are attached to a wireless network.

Resource sharing is the main motivating factor for constructing distributed systems. Resources such as printers, files, web pages or database records are managed by servers of the appropriate type. For example, web servers manage web pages and other web resources. Resources are accessed by clients for example, the clients of web servers are generally called browsers.

The construction of distributed systems produces many challenges:

Heterogeneity: They must be constructed from a variety of different networks, operating systems, computer hardware and programming languages. The Internet communication protocols mask the difference in networks, and middleware can deal with the other differences.

Openness: Distributed systems should be extensible – the first step is to publish the interfaces of the components, but the integration of components written by different programmers is a real challenge.

Security: Encryption can be used to provide adequate protection of shared resources and to keep sensitive information secret when is transmitted in messages over a network. Denial of service attacks are still a problem.

Scalability: A distributed system is scalable if the cost of adding a user is a constant amount in terms of the resources that must be added. The algorithms used to access

shared data should avoid performance bottlenecks and data should be structured hierarchically to get the best access times. Frequently accessed data can be replicated.

Failure handling: Any process, computer or network may fail independently of the others. Therefore each component needs to be aware of the possible ways in which the components its depends on may fail and be designed to deal with each of those failures appropriately.

Concurrency: The presence of multiple users in a distributed system is a source of concurrent requests to its resources. Each resource must be designed to be safe in a concurrent environment.

Transparency: The aim is to make certain aspects of distribution invisible to the application programmer so that they need only be concerned with the design of their particular application. For example, they need not be concerned with its location or the details of how its operations are accessed by other components, or whether it will be replicated or migrated. Even failures of networks and processes can be presented to application programmers in the form of exceptions – but they must be handled.

EXERCISES

1.1 Give five types of hardware resource and five types of data or software resource that can usefully be shared. Give examples of their sharing as it occurs in practice in distributed systems. *pages 2, 7–9*

1.2 How might the clocks in two computers that are linked by a local network be synchronized without reference to an external time source? What factors limit the accuracy of the procedure you have described? How could the clocks in a large number of computers connected by the Internet be synchronized? Discuss the accuracy of that procedure. *page 2*

1.3 A user arrives at a railway station that she has never visited before, carrying a PDA that is capable of wireless networking. Suggest how the user could be provided with information about the local services and amenities at that station, without entering the station's name or attributes. What technical challenges must be overcome? *page 6*

1.4 What are the advantages and disadvantages of HTML, URLs and HTTP as core technologies for information browsing? Are any of these technologies suitable as a basis for client-server computing in general? *page 9*

1.5 Use the World Wide Web as an example to illustrate the concept of resource sharing, client and server.

Resources in the World Wide Web and other services are named by URLs. What do the initials URL denote? Give examples of three different sorts of web resources that can be named by URLs. *page 7*

1.6 Give an example of an HTTP URL.

List the main components of an HTTP URL, stating how their boundaries are denoted

and illustrating each one from your example.

To what extent is an HTTP URL location transparent? *page 7*

1.7 A server program written in one language (for example C++) provides the implementation of a BLOB object that is intended to be accessed by clients that may be written in a different language (for example Java). The client and server computers may have different hardware, but all of them are attached to an internet. Describe the problems due to each of the five aspects of heterogeneity that need to be solved to make it possible for a client object to invoke a method on the server object. *page 16*

1.8 An open distributed system allows new resource sharing services such as the BLOB object in Exercise 1.7 to be added and accessed by a variety of client programs. Discuss in the context of this example, to what extent the needs of openness differ from those of heterogeneity. *page 17*

1.9 Suppose that the operations of the BLOB object are separated into two categories – public operations that are available to all users and protected operations that are available only to certain named users. State all of the problems involved in ensuring that only the named users can use a protected operation. Supposing that access to a protected operation provides information that should not be revealed to all users, what further problems arise? *page 18*

1.10 The INFO service manages a potentially very large set of resources, each of which can be accessed by users throughout the Internet by means of a key (a string name). Discuss an approach to the design of the names of the resources that achieves the minimum loss of performance as the number of resources in the service increases. Suggest how the INFO service can be implemented so as to avoid performance bottlenecks when the number of users becomes very large. *page 19*

1.11 List the three main software components that may fail when a client process invokes a method in a server object, giving an example of a failure in each case. Suggest how the components can be made to tolerate one another's failures. *page 21*

1.12 A server process maintains a shared information object such as the BLOB object of Exercise 1.7. Give arguments for and against allowing the client requests to be executed concurrently by the server. In the case that they are executed concurrently, give an example of possible 'interference' that can occur between the operations of different clients. Suggest how such interference may be prevented. *page 23*

1.13 A service is implemented by several servers. Explain why resources might be transferred between them. Would it be satisfactory for clients to multicast all requests to the group of servers as a way of achieving mobility transparency for clients? *page 23*

The Seven Deadly Sins of Distributed Systems

Steve Muir
Department of Computer Science
Princeton University

Abstract

Applications developed for large-scale heterogeneous environments must address a number of challenges not faced in other networked computer systems such as LAN clusters. We describe some of the problems faced in implementing the PlanetLab Node Manager application and present general guidelines for application developers derived from those problems.

1 Introduction

Developers of distributed applications face a number of challenges particular to their environment, whether it be a homogeneous cluster or a global environment like PlanetLab [6, 2], such as unreliable communication and inconsistent state across the distributed system. Furthermore, personal experiences developing and running applications on workstations and/or servers with LAN interconnects often are not applicable, particularly to problems such as resource allocation and sharing. We present a set of guidelines created in response to our experiences maintaining an infrastructure application for PlanetLab, a large-scale heterogeneous environment.

This paper describes the *seven deadly sins of distributed systems*—seven challenges we encountered in developing *Node Manager*, a key infrastructure component of the PlanetLab environment. Although these challenges are described in the context of a particular application we believe the underlying problems are common to all large-scale heterogeneous environments and thus of interest to a wider audience than just current and future users of PlanetLab.

In Section 2 we briefly describe the PlanetLab environment and Node Manager application in order to set the stage for Section 3's enumeration of the seven deadly sins. After describing each challenge, along with concrete examples of how Node Manager addresses it, we conclude with a summary of general principles applicable to the design and development of distributed applications. Note that this paper focuses on challenges faced by application developers, and so does not directly address the problems faced in administering distributed systems such as PlanetLab e.g., system security, effect of network traffic on external sites.

2 PlanetLab

PlanetLab is a global distributed environment consisting of 400+ nodes connected to the Internet at over 175 geographic locations, a mix of academic institutions and research laboratories. Its global nature provides an extremely diverse network environment that makes implementing a distributed application somewhat challenging.

PlanetLab nodes are IA-32 systems, usually running Intel Xeon or Pentium 4 CPUs with some Pentium III and AMD systems thrown into the mix. Each is centrally administered and configured using a bootable CD that ensures the node boots in a standard manner and can also be rebooted into a minimal *debug* mode; this boot mechanism is not intended to be secure against adversaries with local access but merely to provide a reasonably degree of assurance that nodes can be brought into a known state remotely.

The PlanetLab OS is derived from RedHat Linux 9 with a small number of kernel patches applied; the most significant patches are the standard *vserver* patches to provide namespace isolation between users, and *plkmod*, a kernel module that provides CPU scheduling and network virtualisation. These two components together provide users with an environment that appears essentially to be a complete Linux system, with a restricted root account used to configure that system by each user.

2.1 Node Manager

Node Manager (NM) is the infrastructure subsystem that configures the appropriate user environments—*slivers*—on each node. A user creates a *slice* in the PlanetLab Central (PLC) database, specifying such properties as nodes to be instantiated upon, authorised users and resources allocated to that slice. The contents of that database are

then exported to Node Manager as an XML file which is subsequently used to configure slivers on each node.

Node Manager provides users with prompt creation and deletion of slivers while contending with a large number of problems inherent to the distributed nature of PlanetLab. Our experiences developing and maintaining Node Manager have led us to create a list of the most important challenges to be addressed: we refer to these as the seven deadly sins of distributed systems.

2.2 Related Work

PlanetLab is neither the first nor only distributed environment available to network experimenters and developers of new services but we believe it to be both larger and more heterogeneous than earlier such projects. The University of Utah's *Emulab* [8] project provides a large cluster environment for researchers but lacks the heterogeneous network environment of PlanetLab; of course, for some applications that may actually be preferable. MIT's *Resilient Overlay Network (RON)* [1] testbed provides a similar network environment to PlanetLab but on a much smaller scale. Finally, the *Google file system* [3], a cluster-based filesystem, addresses many similar challenges to those encountered in PlanetLab even though the environment is much more homogeneous.

3 The Seven Deadly Sins

For each of the 'sins' listed below we will describe the nature of the problem along with the solutions adopted in Node Manager. While many of these problems are well-known and in some cases the subjects of vast amounts of existing research, we believe that the practical context in which we encountered them helps motivate the applicability of their solutions to PlanetLab and similar distributed systems.

1. Networks are unreliable in the worst possible way
2. DNS does not make for a good naming system
3. Local clocks are inaccurate and unreliable
4. Large-scale systems always have inconsistencies
5. Improbable events occur frequently in large systems
6. Overutilisation is the steady-state condition
7. Limited system transparency hampers debugging

Note that this list is far from complete, but reflects the set of challenges we faced implementing Node Manager, particularly where the problem or its effects were surprising.

3.1 Large Heterogeneous Networks are Fundamentally Unreliable

The single biggest challenge facing architects of distributed applications is the simple fact that networks are unreliable. Although the specifications of IP, TCP, UDP, etc., all make clear the various forms this unreliability can take, it is easy for application developers using high-level languages and systems to forget, or fail to appreciate, the practical implications of these problems for their applications. Often, this unreliability does not take a convenient form from the architect's perspective—the fate of network packets is not simply sent or discarded, they may also be delayed, duplicated and/or corrupted. Even 'reliable' protocols such as TCP still leave the designer to deal with such problems as highly variable latency and bandwidth and unexpected connection termination.

The implications of these problems for distributed application development are significant. Some are obvious—what should the application do if a connection is refused—whereas others are more subtle—what should the application do if a small file takes 24 hours to download, or if a remote procedure call (RPC) is interrupted?

Perhaps the foremost rule in handling network problems is that all possible errors should be handled gracefully—as the 5th problem described below says, it's the error condition that isn't handled that will be the one that occurs sooner or later when an application runs 24/7 on hundreds of nodes. Large, geographically distributed networks *will* exhibit all kinds of network failure modes, including the obscure ones that never occur on local networks: packet reordering, duplicate packets arriving at a destination.

Secondly, many transient failures occurring within the network, e.g., packet loss, are successfully hidden from applications by the network stack when in practice the application needs to be aware of them. For example, we frequently found that poor connectivity to our central servers would lead to file downloads, even of small files, taking many hours. This led to two problems: first, the application becomes tied up in file download, thus being unable to perform other operations or respond to new service requests; and second, the received data is often stale by the time download is completed. The first problem can be addressed in several ways, but we use two complementary solutions in Node Manager: multithreading (or asynchronous I/O) is used to perform long latency operations while responding to other events, and timeouts (if chosen appropriately) are effective in terminating operations that do not complete promptly. Stale data can be identified using timestamps, although validating such a timestamp has its own challenges (see Section 3.3).

RPC operations that terminate abnormally present another class of problem. The client typically has little or no information about the state that the server was in when

the operation terminated, so determining how to handle this problem depends to a great deal upon the nature of the operation. Transaction processing mechanisms are one solution but may be too heavyweight for the rapid prototyping of new network services that is the focus of PlanetLab; one solution adopted in Node Manager is for the server to maintain only minimal internal consistency properties such that an aborted operation does not adversely impact subsequent requests, and then have external applications periodically verify that the server state is globally consistent.

For example, Node Manager provides a pair of operations—*acquire* and *bind*—that are used together to acquire a set of resources, returning a handle to the caller, then use that handle to bind the resource set to a particular slice. If the acquire operation fails abnormally i.e., due to an RPC mechanism error rather than an error returned by Node Manager, the client has no information about whether the resource set was allocated. Since the client did not receive the resource set handle it has no way of determining that information, so it retries the request; in Node Manager itself we periodically cleanup allocated resource sets that have not been bound.

Finally, distributed applications may have to deal with interference from other network users. Random connections from external networks, such as port-scanning of network interfaces, are not uncommon, so applications providing network services must be able to handle such invalid connection attempts gracefully. In particular, higher-level libraries may have internal error handling designed for simple client-server applications that is inappropriate in the target environment.

3.2 DNS Names Make Poor Node Identifiers

A key challenge in any distributed system is naming of entities in the system [7]. We focus only on the challenge of naming nodes in the system i.e., assigning a name N to a node such that N unambiguously refers to that node and that node only, and the node can reliably determine, in the face of unreliable communication, that its name is N.

DNS names are appealing candidates as a solution for several reasons: they're simple, they're well understood by developers and users, and there's a large amount of infrastructure and tools to support use of them. Unfortunately, DNS names suffer from both *ambiguity* and *instability*—DNS names may not be unique and are not stable over long periods of time. These problems arise due to several root causes:

- Human errors: DNS names are assigned by system administrators, so there are often mistakes made: the same name assigned to multiple nodes or vice versa, reverse lookups not matching forward lookups.
- Network reorganisation: sometimes sites change their internal network addressing, thus requiring that DNS records be updated; hostnames are even occasionally changed to more 'user-friendly' variants e.g., `nwu.edu` changed to `northwestern.edu`.
- Infrastructure failures: DNS servers may fail completely or be overloaded, thus forcing requests to be sent to a secondary server.
- Network asymmetry: nodes may have internal DNS names—within their local institution's network infrastructure—that differ from their external name, perhaps due to NAT; this is particularly problematic in the face of infrastructure failures.
- Non-static addresses and multihoming: both can introduce further complexity into node naming if the node derives its name from its IP address.

The consequences of these problems are twofold: external entities may not be able to identify and access a node through its DNS name, and a node that attempts to determine its own DNS name from its network configuration cannot reliably do so. For example, some PlanetLab nodes had CNAME (alias) records that were provided only by certain nameservers—when the primary nameserver failed and the node asked one of these servers for its name (using its IP address) it got back a different result than it would have gotten from the primary nameserver. Similarly, when the reverse mappings for two distinct IP addresses were erroneously associated with the same name we found that two nodes believed they had the same name.

An obvious alternative to using DNS for naming nodes is to use IP addresses, but unfortunately they aren't much better—they do occasionally change, even on 'static' networks e.g., due to local network reconfiguration, and the inherent non-human readable nature of IP addresses always leads to the temptation to convert to DNS names for various purposes, thus introducing DNS-related problems, such as high latency for lookups and reverse lookups, into unrelated code e.g., formatting debug messages.

In PlanetLab we adopted unique numeric IDs as node names, with those IDs being generated by our centralised database whenever a node is installed; a node ID is associated with the MAC address of the primary network adapter, so node IDs do occasionally change, but we have found this scheme to be more reliable than either DNS names or IP addresses for naming nodes.

3.3 Local Clocks are Unreliable and Untrustworthy

A second problem often encountered in distributed systems is maintaining a globally consistent notion of time [4]. Specific requirements in PlanetLab that have

proven problematic include determining whether an externally generated timestamp is in the past or future, and monitoring whether a particular subsystem is live or dead. Of course, as a testbed for network experiments it is also imperative that PlanetLab provide a reliable clock for measurement purposes.

The root cause of time-related problems is the fact that local clocks are sometimes grossly inaccurate: we commonly observed timer interrupt frequency errors of 2–3% on some nodes, most likely due to bad hardware or interrupts being lost due to kernel bugs, and in the most extreme cases we observed nodes 'losing' about 10% of their timer interrupts over a 1-hour period.

NTP [5] really helps but some sites block NTP ports, and occasionally on heavily loaded systems, which almost all PlanetLab nodes are (see Section 3.6), NTP doesn't run frequently enough to compensate for massive drift. While ideally kernel bugs would be identified and fixed we have found that short-term solutions, such as adjusting the timer period to compensate for lost interrupts, can make a significant difference. Finally, some applications e.g., local measurements, don't actually require a globally correct clock, just a timer of known period for which a facility like the IA-32 timestamp counter is perfectly adequate.

The magnitude of clock errors that are considered reasonable places limits on the granularity at which timestamps are useful. For example, once we realised that the NTP service on our nodes was frequently making adjustments of tens of seconds every 15–20 minutes it becomes impractical to allocate resources with expiration times in the minute range—an NTP adjustment can advance the local clock so far that a resource just allocated becomes expired immediately. Furthermore, although NTP limits the size of its adjustments (usually to ± 1000 seconds), an administrator who sees that a node's clock has deviated far from the correct value will often manually reset the clock, sometimes by as much as several days—applications must therefore be designed and implemented to detect gross changes in local time and respond gracefully.

Similarly, external monitoring of nodes e.g., to detect when an event was last recorded, is unreliable if the timestamp recorded with the event is not known to be consistent with the monitor's notion of time. The solution depends upon the particular type of monitoring: for live services we have found that having a simple 'ping' facility in the application that can be exercised by the monitor is preferable, while the accuracy of timestamps in, say, log files can be increased somewhat by considering the current difference between the monitor's clock and the monitored node's local clock, if that difference is assumed to be representative, or used to calculate the appropriate value.

3.4 Inconsistent Node Configuration is the Norm

It's hard to maintain a consistent node configuration across a distributed system like PlanetLab. Typically a significant fraction of nodes will not have all the latest versions of software packages and the most up-to-date configuration information, usually because of network outages that prevent the auto-configuration service from downloading new files. In addition to leaving nodes exposed to security vulnerabilities if recent updates have not been applied, application designers must explicitly handle inconsistencies between multiple versions of data files and software packages; we focus only on the latter problem.

The biggest effect of this global inconsistency is that system administrators cannot make drastic changes to data distributed to the nodes without taking into account the ability of old versions of node software to handle the new data (and corresponding metadata e.g., file formats). This problem is exacerbated by the fact that software and configuration updates may not be well-ordered: a change to a configuration file may be applied to a node even though a prior change to the corresponding software package has not been applied.

For example: in PlanetLab we maintain a centralised database of user *slices*—high-level virtual machines—that are distributed across PlanetLab nodes. The contents of this database are exported to nodes via an XML file retrieved using HTTP (over SSL). When changes are made to the database and/or the subset of that database exported in that XML file we have to consider how our changes will affect nodes running out-of-date versions of the Node Manager software—if the format of the slice description changes will NM fail to recognise that a slice should be instantiated on the local node, and furthermore delete it from that node if already instantiated?

Applications such as NM can be made more robust against unexpected data format changes to a certain degree, and it is often also possible to incorporate failsafe behaviour, so that, say, the sudden disappearance of all slices from the XML file, is recognised as most likely being due to a major format change. The most obvious way to prevent unexpected behaviour due to signficant changes is to associate version numbers with file formats, protocols, APIs, etc., so that large changes can be easily detected. But it is generally not possible to completely isolate data publishers—those entities that push data out to the nodes—from the need to consider the effects of data format changes upon application software.

3.5 There's No Such Thing as "One-in-a-Million"

In a distributed system with hundreds of nodes running 24/7, even the most improbable events start to occur on a not-too-infrequent basis. It's no longer acceptable to ignore corner cases that probably never occur—those cor-

ner cases will occur and will break your application.

For example: the Linux *logrotate* utility rotates log files periodically. It can be configured to send a SIGHUP signal to the daemon writing a log file when the log file is rotated, so that the daemon can close its file handle and reopen it on the new file. If the daemon happens to be executing a system call when the signal is received that signal call will be terminated with an EINTR error which must be handled correctly by the daemon. If logs are rotated only once a day or even once a week, and the daemon only spends 1% of its time executing system calls, then it becomes very tempting to ignore the possibility that the signal will be received at the most inopportune moment; unfortunately, as we discovered, this possibility will actually happen with a non-negligible frequency.

A similar problem arises with filesystem corruption due to nodes being rebooted without being shutdown properly, an event that also occurs much more frequently than one's intuition leads one to believe. Whereas a user running Linux on their desktop only very infrequently finds their system rebooted involuntarily e.g., due to power outages, in a distributed system located at hundreds of sites it is not uncommon for there to be one or more such outages every week. Consequently we find that instances of filesystem corruption are not uncommon, particularly since our nodes are frequently very heavily loaded and so the probability of there being inconsistent metadata when such an outage occurs is high.

Hence application developers must not cut corners when handling error conditions, and must be careful that their own personal experiences of using one or two computers at one or two locations do not lead to incorrect assumptions of whether unusual events will occur.

3.6 No PlanetLab Node is Under-Utilised

We observe that even with an increasing number of PlanetLab nodes, each one is still more or less fully utilised at all times. CPU utilisation is typically 100%, with twenty or more independent slices concurrently active within even short time periods (on the order of a few minutes). Hence it is not uncommon to find several hundreds of processes in existence at any instant in time, and load averages i.e., number of runnable processes, in the 5–10 range are normal. This is yet another example where user experience running an application on a single-user workstation doesn't provide good intuition as to how that application will behave when deployed on PlanetLab.

The most obvious effect of this over-utilisation is that applications typically run much slower on PlanetLab, since they only receive a fraction of the CPU time available. While this is often not in itself a problem, it can have unforeseen implications for functions that make implicit assumptions about relative timing of operations. For example, the PLC agent component of Node Manager uses a pair of RPC operations to create new slices: in an unloaded system the first operation—acquiring a resource handle—completes very quickly, so the resource corresponding to the new handle is likely to be available for use by the second RPC operation. When the system is heavily loaded, the latency of the first operation can increase up to several minutes or more, so the state of the system when the second RPC is invoked may have changed significantly and the resource referred to by the handle may no longer be available.

The inherent global nature of such resource overconsumption means that a system-wide solution is often most effective; in PlanetLab we implemented a proportional share CPU scheduler to guarantee that each slice gets an equal (or weighted) share of the CPU within every time period (typically a few seconds). However, it is still prudent for applications to anticipate the effects of heavy load: fortunately many of the enhancements that are appropriate are similar to those concerned with inaccuracies in local clocks e.g., not allocating resources with very short timeouts. Another measure adopted by Node Manager to handle heavy load is to detect when an operation takes too long (obviously defined in an application-specific manner) and build a degree of leniency into the system e.g., extend resource expiration times in such cases.

3.7 Limited System Transparency Hampers Debugging

A further difference between developing an application on a single-user workstation and deploying it on a distributed system like PlanetLab is that one often does not have complete access to system nodes in the latter environment. While some distributed systems only provided users with restricted accounts, in other cases the lack of transparency is due to virtualisation that prevents the illusion of full access while hiding other system activity from the user.

For example, in PlanetLab we allocate user slices as environments within which services and experiments are run, each slice being an isolated VM that gives the appearance, albeit only if one doesn't look *too* closely, of each user having unrestricted access to a Linux system; each user has a restricted root account within their slice that can be used to run standard system configuration tools such as RedHat's rpm or Debian's dpkg.

Unfortunately the shared nature of PlanetLab nodes combined with the limited view of the system available to each user can make it more challenging for individual users to debug their applications. Similarly to the previous point, this problem is best addressed by system-wide measures, such as providing new tools that can be used by slice users to obtain the required debugging and status information. One example is the PlanetLab *SliceStat* ser-

vice that exports aggregate process information for each slice i.e., the total amount of memory, CPU time and network bandwidth consumed by all processes within a slice.

From the individual user's perspective, this lack of visibility into the distributed system emphasises the importance of debugging the application locally as thoroughly as possible before deploying to PlanetLab, ideally under simulated conditions that closely resemble those of the distributed environment e.g., heavy CPU load. Reporting as much information as is available to the application rather than relying upon traditional system monitoring tools—dumping received packets in a readable manner rather than using an auxiliary `tcpdump`, say—also pays dividends.

4 Conclusions

Application developers who are designing and implementing new distributed applications, or porting an existing application, have a number of challenges inherent to the distributed environment to address. Our experiences implementing the Node Manager component of the PlanetLab infrastructure led us to develop a list of the seven most important of these challenges. From the solutions adopted to address these challenges a smaller set of general guidelines emerged:

- Many assumptions made in non-distributed applications are not valid in large-scale and/or heterogeneous environments.
- Distributed applications must gracefully handle a broad variety of corner-case failure modes that are often ignored in the non-distributed environment.
- Resource management in the distributed environment is significantly different from the non-distributed case.
- Even local operations can behave radically differently in a system that is heavily over-utilised.

By following these guidelines in the implementation of Node Manager we have successfully increased the robustness of the distributed application and helped make PlanetLab a more reliable environment for deploying new network services.

References

[1] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient Overlay Networks. In *Proc. 18th SOSP* (Banff, Alberta, Canada, Oct 2001), pp. 131–145.

[2] CHUN, B., ET AL. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review 33*, 3 (Jul 2003).

[3] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[4] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (Jul 1978), 558–565.

[5] MILLS, D. L. Internet time synchronization: The Network Time Protocol. *Internet Req. for Cmts.* (Oct. 1989).

[6] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets-I* (Princeton, NJ, Oct 2002).

[7] WATSON, R. W. Identifiers (Naming) in Distributed Systems. In *Distributed Systems—Architecture and Implementation, An Advanced Course*. Springer-Verlag, 1981, pp. 191–210.

[8] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5th OSDI* (Boston, MA, Dec 2002), pp. 255–270.